# RT0704

## Programmations des applications en réseaux

# Projet

Paul Legendre – Master ASR Projet RT0704 – 2019/2020 – URCA Faculté des sciences

# Sommaire

# 1. Introduction :

L’objectif du projet est de mettre en place une plateforme de code à la demande. Cette infrastructure utilisera plusieurs technologies comme Linux, RabbitMQ, Docker et Python.
Le but est de permettre à un client d’envoyer un traitement vers un exécutant qui aura pour but de le diviser en plusieurs tâches.

# 2. Technologies utilisées :

Système d’exploitation : Linux Ubuntu 18,04
Serveur de file de message : RabbitMQ
Serveur web : Flask (ubuntu 18.04)
OS worker : (ubuntu 18.04)
Technologie de virtualisation : Docker
Gestionnaire de dépôts : Git & Github
Langage de développement : Python
Visual Code
Putty (accès ssh)

# 3. Organisation de l’infrastructure :

La machine hôte sous Ubuntu 18.04 lancera :

- le code MAIN.py (situé dans */home/test/env/projet/MAIN.py)*
- le code commanditaire qui sera un thread agira en tant que client
- le code exécutant qui sera thread agira en temps qu’exécutant
- les containers Worker et Flask seront sous Ubuntu 18.04
- le container RabbitMQ sera le container officiel docker RabbitMQ

***Explication des dossiers :***

*env/TP1 : contient mes travaux du TP1*
*env/projet :contient les ressources du projet TP1 partie 2*

**Utilisation et installation :**

Projet partie 1:
1. copier le dossier env situé dans *« /home/test/ »*
*2. activer l’environnement python « source env/bin/activate »*
*3. lancer le script MAIN.py « python3 MAIN.py »*

*TP partie 0:*
*1. dossier zmapp lancer le script run.py*
*2. dossier git_tp contient les test GIT effectués en python*
*3. dossier github contient les test GITHUB effectués en python (pygithub)*
*4. dossier zmapp_cli contient un interface web pour intéragir avec RabbitMQ*

# 4. Vue d’ensemble de l’infrastructure :

schéma UML de l’infrastructure complète

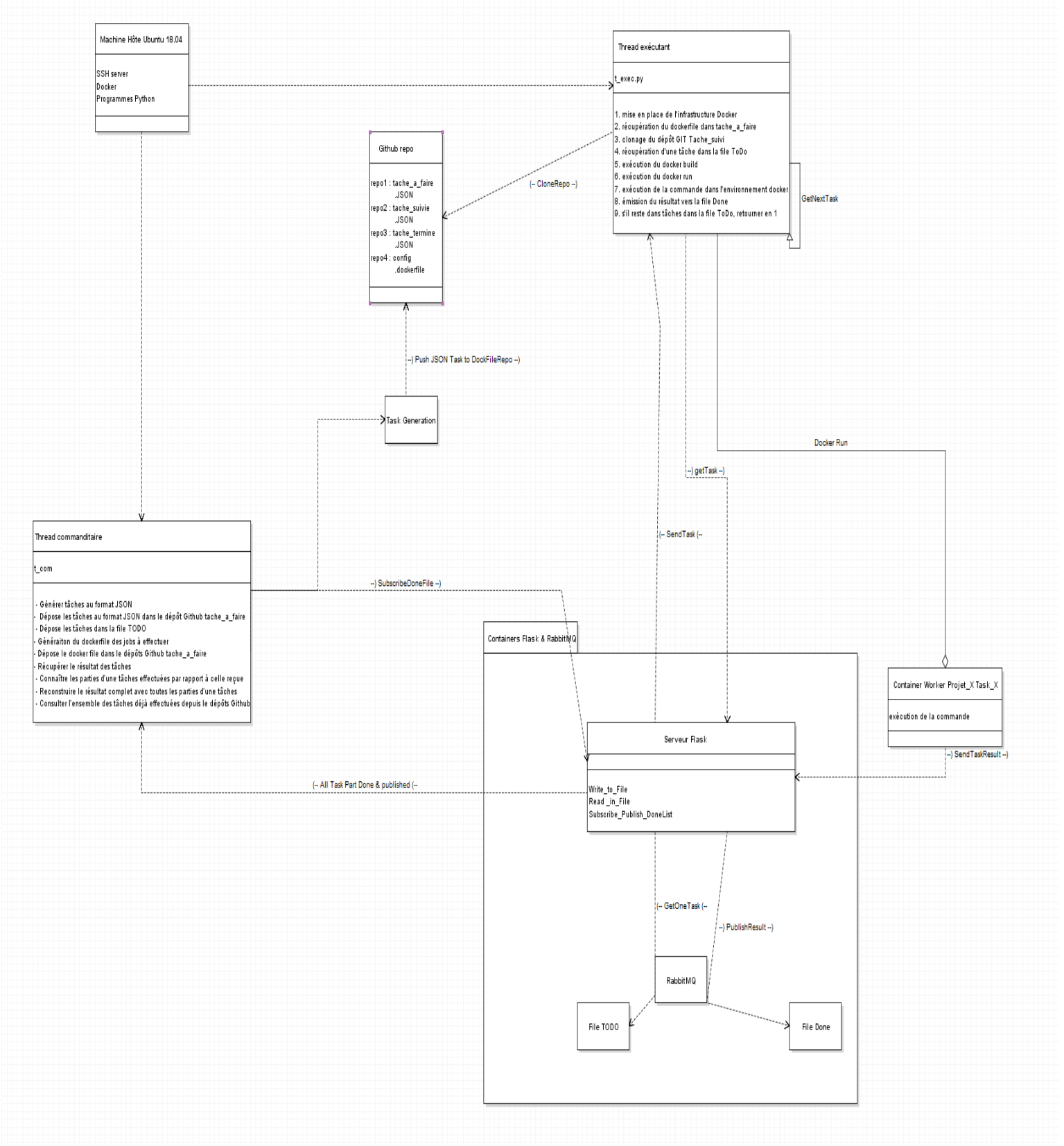
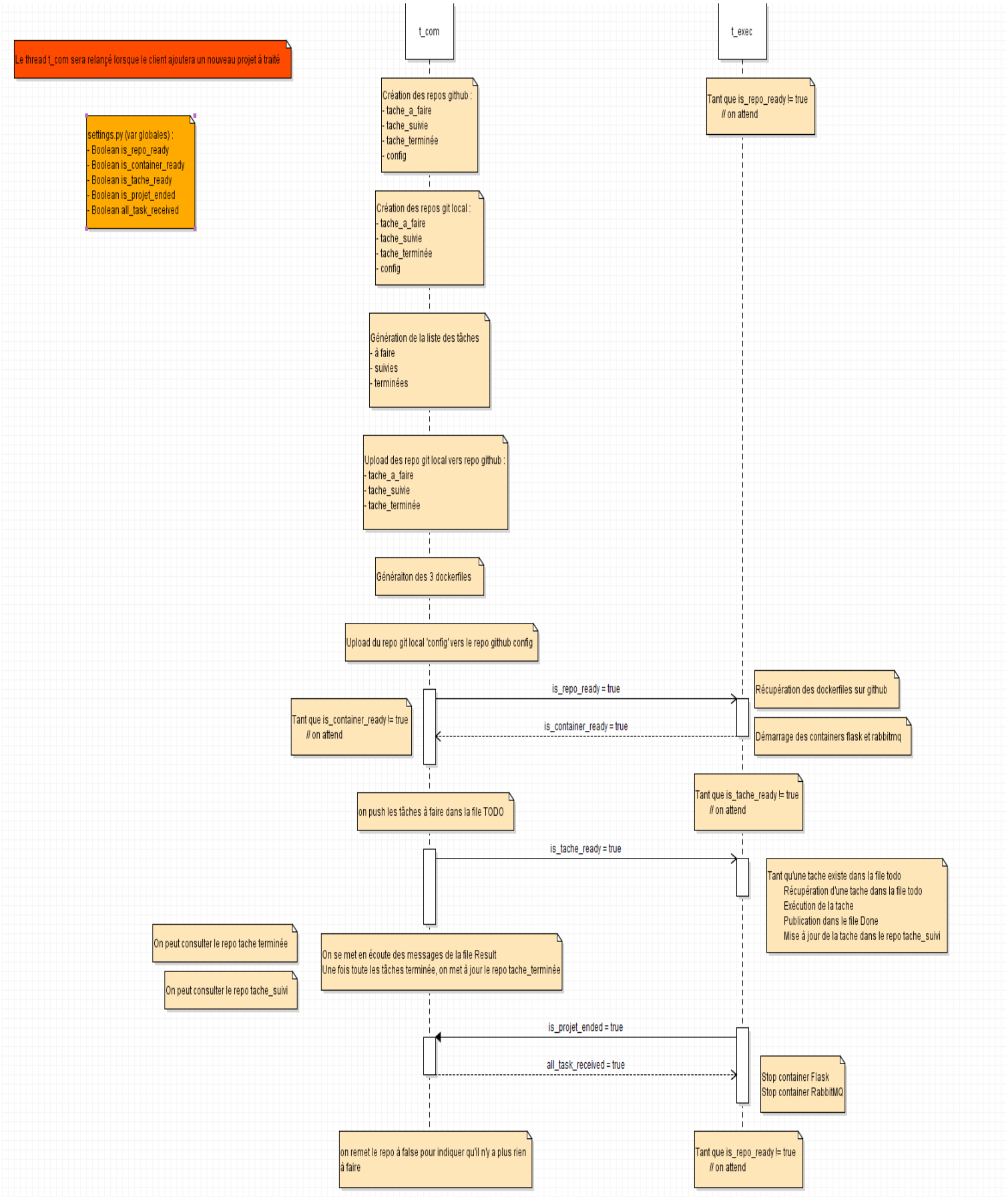


Schéma de comment interagissement les 2 threads du script MAIN.py



## 5. Projets et Tâches :

**Quoi ?**

Un projet est un ensemble de tâche représentant les traitement que le commanditaire veut effectuer.
On pourrait imaginer qu'il a accès à un interface web afin de créer un projet de tâche et un web service permettant de le générer au format JSON.
Dans le cadre de ce projet, on le générera grâce à python au format JSON.
Cette liste de tâche sera donc générée par projet et comportera une nomination « 1_… » faisant référence au projet 1.

**Configuration** :

Le traitement imposé de chaque tâche devrait être le résultat de l'algorithme pour n-dames.
Afin de mettre en place mon projet, j'ai choisi que chacune des tâches d'un projet aura pour but d'afficher une phrase avec la commande echo généré dans une boucle.
Cette phrase portera le nom du projet et son numéro de tâche dans son projet correspondant.
Le résultat de la tâche effectuée et reconstruite sera donc l'affichage d'un ensemble de phrase.
En annexe se trouve l'algorithme des n-dames que je n'ai pas eu le temps d'implémenter.

**Structure :**

On utilisera donc le format JSON pour les tâches d'un projet:

```
[
    {
        "cmd": "echo ",
        "id_projet": 0,
        "id_task": "0",
        "phrase": "la tache numero 0 dit bonjour au projet numero 0",
        "url": "/home/test/env/projet/mgit_repo/tache_a_faire"
    },
    {
        "cmd": "echo ",
        "id_projet": 0,
        "id_task": "1",
        "phrase": "la tache numero 1 dit bonjour au projet numero 0",
        "url": "/home/test/env/projet/mgit_repo/tache_a_faire"
    }
]
```

## 6. Code Commanditaire :

**Rôles :**
Le commanditaire agira donc comme un client. Il initiera les tâches à effectuer.

**Configuration :**

C’est un thread pyhton « t_com » lancé depuis le script MAIN.py, ses traitements respectent ceux imposés par le sujet.

**Étapes de démarrage et de fonctionnement :**

X. Lancer par Main.py Thread « t_com »
    X. Génération des 4 repos github
    X. Génération des 4 repos git
    X. Génération de la liste des tâches, les listes de suivie des tâches
    X. Push de ces 4 repos sur github
    X. Génération des Dokerfile
    X. Push des tâches dans la file TODO
    X. Souscription à la file Done
    X. Gestion des job terminées par tâches
    X. Mise à jour du dépôt « taches_suivi »
    X. Reconstruction d’un projet avec toutes les tâches terminées

## Code :

```python
class t_com(Thread):

    def __init__(self, name):
        Thread.__init__(self)
        self.name = name

    def run(self):
        # définition des variables globales partagées entre les threads
        global shared_data
        global is_beginning
        global is_repo_ready
        global is_container_ready
        global is_tache_ready
        global is_projet_ended
        global all_task_received

        print("\n\n##################################################")
        print("## ThreadName started"+self.name+" ## ")
        print("##################################################\n\n")

        is_beginning = True
        #print("recuperation de la valeur de la variable globale= "+str(shared_data))
        #with verrou:
            #shared_data = shared_data +1
            #print("shared_data modif= "+str(shared_data))

        # variables globales (settings)
        print_boolean_follow()

        # on génère les 4 repos github
        create_remote_repo(user_github, repo_tache_a_faire)
        create_remote_repo(user_github, repo_tache_suivi)
        create_remote_repo(user_github, repo_tache_termine)
        create_remote_repo(user_github, repo_config)

        # on génère les 4 repos git local
        create_repo(base_path_repo, repo_tache_a_faire)
        create_repo(base_path_repo, repo_tache_suivi)
        create_repo(base_path_repo, repo_tache_termine)
        create_repo(base_path_repo, repo_config)

        ## on devrait boucler içi pour générer de nouvelle boucle de traitement

        # on génère la liste de tache + liste de suivi de tache + liste de tache terminée
        list_task = generate_task(current_id_projet, "la phrase", task_nb, base_path_repo, repo_tache_a_faire)
        json_list_task = json.dumps(list_task, sort_keys=True, indent=4, separators=(',', ': '))
        # on génère liste de suivi de tache + liste de tache terminée
        list_taskS = generate_task_suivi(current_id_projet, list_task)
        json_list_taskS = json.dumps(list_taskS, sort_keys=True, indent=4, separators=(',', ': '))
        # on génère liste de tache terminée
        json_ProjectT = generate_task_terminee(current_id_projet, list_taskS)

        # on push la liste de tâche à faire sur le repot git local
        push_task(json_list_task, current_id_projet, base_path, base_path_repo, repo_tache_a_faire, user_github)
        # on push la liste de tâche suivi  sur le repot git local
        push_taskS(json_list_taskS, current_id_projet, base_path, base_path_repo, repo_tache_suivi, user_github)
        # on push la liste de tâche suivi  sur le repot git local
        push_ProjectT(json_ProjectT, current_id_projet, base_path, base_path_repo, repo_tache_termine, user_github)

        # on push les 3 différents repos vers les repos github correspondant
        push_github(base_path_repo, repo_tache_a_faire, base_url_remote_repo, branch, user_github)
        push_github(base_path_repo, repo_tache_suivi, base_url_remote_repo, branch, user_github)
        push_github(base_path_repo, repo_tache_termine, base_url_remote_repo, branch, user_github)
```

```
# on génère le dockerfile worker dans le dépôts config/worker
# on génère le dockerfile flask dans le dépôts config/flask
# on génère le dockerfile file dans le dépôts config/file
# on push le repo local config vers le repo github config

# on informe l'éxécutant que les repos sont prêts et que le traitement peut commencer
print("\n## ThreadName "+self.name+" ## \nis_repo_ready OK \nis_beginning = True\n");
is_repo_ready = True

while is_container_ready == False :
    print_period("\n\n## ThreadName "+self.name+" ## \nen attente du demarrage des containers de la part de l'executant\n");

# on push les tâches à faire dans la file de messages TODO
push_task_todo()

## on informe l'exécutant que les tâches à faire ont été placées dans la file de message
is_tache_ready = True
# on s'abonne à la File Done

while is_projet_ended == False :
    print_period("\n## ThreadName "+self.name+" ## \ntant que toute les taches du projet n'ont pas ete traitees\n");
    #print_boolean_follow()
    # A chaque résultat reçue :
    # on stockera le résultat de la tâche dans la list_tache_suivi
    # on affichera l'ensemble des tâches déjà effectuées par rapport à celles reçues

# si toutes les tâches d'un projet sont reçues alors on reconstruira le résultat que l'on stockera dans la list_tache_termine
# le résultat d'une tâche terminée sera stockée dans le dépot github tâche terminée

# on informe l'exécutant que le résultat de toutes les tâches du projet ont été reçues
all_task_received = True

# on peut consulter à tout moment les tâches_suivi en affichant le repo tache_suivi
# on peut consulter à tout moment les tâches_termine en affichant le repo tache_termine

# on remet l'état des taches à False
time.sleep(10)
# on informe l'exécutant de ne pas commencer une nouvelle boucle de traitement
is_repo_ready = False
is_tache_ready = False
all_task_received = False
is_beginning = False
```

# 7. Code Exécutant :

**Rôles :**

Le code Exécutant est donc le thread qui aura pour but la prise en main du traitement et sa répartition.Il s'occupera de lancer par exemple un worker pour chaque tâche récupérée dans la file TODO.

**Configuration :**

C'est un thread python lancé depuis le script MAIN.py, ses traitements respectent ceux imposés par le sujet.

**Étapes de démarrage et de fonctionnement :**

X. Lancer par Main.py Thread « t_exec »
    X. Clonage du dépôts GIT « taches_a_faire »
    X. Build du dockerfile RabbitMQ et Run du container associé à cette image
    X. Build du dockerfile Flask et Run du container associé à cette image
    X. Création des files DONE et TODO par web service Flask
    X. Lancer des workers pour chaque message récupérés
    X. Mise à jour du dépôt local etgithub tache_suivi

**Code :**

```
class t_exec(Thread):

    def __init__(self, name):
        Thread.__init__(self)
        self.name = name

    def run(self):
        # définition des variables globales partagées entre les threads
        global shared_data
        global is_beginning
        global is_repo_ready
        global is_container_ready
        global is_tache_ready
        global is_projet_ended
        global all_task_received

        print("\n\n#####################################################")
        print("## ThreadName started"+self.name+" ## ")
        print("#####################################################\n\n")
        #print("recuperation de la valeur de la variable globale= "+str(shared_data))
        #with verrou:
            #shared_data = shared_data +1
            #print("shared_data modif= "+str(shared_data))
        while is_beginning == True:
            while is_repo_ready != True:
                print_period("\n## ThreadName "+self.name+" ## \nen attente de l'initialisation des repots de la part du commanditaire\n\n");

            # le commanditaire a initialisé les repos

            # on clone localement le dépôts Github tâche_suivi

            # on récupère le dockerfile rabbitmq dans le dépôts Github config/file
            ## on build l'image correspondant à ce dockerfile
            build_dockerfile("rabbitmq_srv")
            ## on démarre le docker RabbitMQ
            start_container("rabbitmq_srv")

            # on récupère le dockerfile flask serveur dnas le dépôts Github config/flask
            ## on build l'image correspondant à ce dockerfile
            build_dockerfile("flask_srv")
            ## on démarre le docker
            start_container("flask_srv")

            # on récupère le dockerfile worker dans le dépôts Github config/worker
            ## on build l'image correspondant à ce dockerfile
            build_dockerfile("worker")

            # on doit savoir quand le docker flask et rabbitmq sont démarrés
            while ping("172.17.0.2") != True or ping("172.17.0.3") != True :
                pr = str(ping("172.17.0.2"))
                pf = str(ping("172.17.0.3"))
                print_lperiod("\n## ThreadName "+self.name+" ## \nen attente du demarrage du container rabbitmq et du container flask\nrabbitmq up= "+pr+"\nflask_srv up= "+pf+"\n");

            # on crée les files de message DONE et TODO par l'url du serveur flask
            if create_queue('TODO') == True and create_queue('DONE') == True :
                print("file de message TOTO et DONE created")
            else :
                print("ERREUR files de message non created")

            # on informe le commanditaire que les containers sont prêts
            print("\n## ThreadName "+self.name+" ## \is_container_read OK \n");
            is_container_ready = True
```

```
            while is_tache_ready != True:
                print_period("\n## ThreadName "+self.name+" ## \nen attente de taches dans la file TODO\n\n");

            # le commanditaire a placé des tâches dans la file TODO
            ## on démarre le traitement des tâches

            ##### Tant qu'il y a des messages dans la file todo ( == getTask != none)
            is_empty_file = False
            while : is_empty_file != False
                # on récupère une tache dans la file Todo
                ## request to Flask serveur /getTask
                msg = get_msg('TODO')
                id_projet = msg['id_projet']
                id_task = msg('id_task')
                cmd = msg['cmd']
                phrase = msg['phrase']

                # on lance un worker avec les attributs(id_projet, id_task, id_projet, cmd, phrase) de cette tâche récupéré dans le message par /getTask
                ## on démarre le docker
                start_container(id_projet, id_task, cmd, phrase)

                # on met à jour le dépôts local tache_suivi avec le resultat et l'état ended à TRUE de la tâche terminée
                ## on push cette modification sur le dépôts Github tache_suivi
            ####

            # il n'y a plus de tâches dans la file TODO
            # on avertit le commanditaire que toutes les tâches du projet ont été traité
            is_projet_ended = True

            while all_task_received == False :
                print_period("\n## ThreadName "+self.name+" ## \nen attente de la confirmation du commanditaire\n\n");
                print_period("\n## ThreadName "+self.name)
                #print_boolean_follow()
                print("########")

            # le commanditaire a confirmé que les résultats de toutes les tâches ont été reçues
            ## on stop le container Flask
            ## on stop le container RabbitMQ
            ## on remet l'état des containers à False
            is_projet_ended = False
            is_container_ready = False

            # on boucle sur le traitement
            # le commanditaire devra relancer le thread commanditaire pour relancer le traitement exécutant
            while is_beginning == False :
                print_period("\n## ThreadName "+self.name+" ## \nen attente du commanditaire pour une nouvelle boucle de traitement\n\n");
```

# 8. Le(s) Worker(s) :

**Quoi ?**
Le Worker est donc container lancé par le code exécutant, il aura pour but de traiter un message de la file Todo.

Il effectuera donc le traitement d'une tâche d'un projet.

**Rôles :**

- Émission du résultat

**Configuration :**

- IP : 172.17.0.4 jusqu'à 172.17.0.X
- Container Ubuntu 18.04
- Lancement d'une commande shell « echo phrase »

**Étapes de démarrage et de fonctionnement :**

- docker build ./ -t worker
- docker run --name worker worker

**Dockerfile :**

```
############################### Worker DockerFile Ubuntu ###################################################
# ip_adress: 172.17.0.4+ / gateway : 172.17.0.1 (mode bridge)
# exécutera le traitement d'une tâche d'un projet
# Plusieurs informations devront lui être communiquée
# - numéro de projet
# - numéro de la tâche
# - commande spécifié de la tâche (echo)
# - argument de la tâche(phrase)
# - dossier partagée pour ce docker
# - nom du script.sh lancé au démarrage
# - nom de l'image
# - nom du tag de version
# - alias de l'image
# - addresse ip du serveur flask 'http://172.18.10.10:51'
# - endpoint du service /send_result
# devra renvoyer la phrase affichée
############################################################################################################

# Définie l'image qui sera utilisé et lui donne un nom
FROM ubuntu:18.04 AS worker

# exécuter des commandes dans votre conteneur
RUN apt-get update -y
# RUN apt-get install -y ping
RUN apt-get install -y iputils-ping
# installation de pyhton
RUN apt-get install -y python3-pip python3-dev build-essential

# Créer un point de montage avec un nom spécifique
# permet d'indiquer quel répertoire vous voulez partager avec l'hôte
# le répertoire de l'hôte qui sera partagé doit être spécifié au lancement du container
VOLUME /dir_worker

# WORKDIR : défini le répertoire pour RUN, CMD, ENTRYPOINT, COPY, ADD
# permet de modifier le répertoire courant (équivalent de cd)
# si le dossier n'existe pas alors il sera crée automatiquement
WORKDIR /dir_worker

# COPY : copie un nouveau fichier, dossier ou données distante et de les ajouter au système de fichier de <dest>
COPY /worker_entry.sh /dir_worker/
COPY /dojob.py /dir_worker/

# ENTRYPOINT: Permet de configurer le container afin qu'il lance un script
# ENTRYPOINT ["/dir_worker/worker_entry.sh"]
ENTRYPOINT ["python3"]
CMD ["/dir_worker/dojob.py"]

############################################################################################################
```

# 9. RabbitMQ :

**Quoi ?**

Les 2 files de messages RabbitMQ sont « TODO » et « DONE ».

« TODO » permet de stocker les tâches et  se raconnsomés par le code exécutant
Chaque message contiendra les informations associées à cette tâche :
- id projet - id tache - commande à exécuter - ressource liée à la commande

« DONE » permet de stocker les tâches terminées et sera consommés par le code commanditaire.

**Rôles :**

- gestion des files « DONE » et « TODO » ( émission et réception )

**Configuration :**

- IP : 172.17.0.2 (1er container à être lancé)
- Container Ubuntu 18.04
- un forward de port a été configuré sur virtual box du port hôte 8888 vers le 9999 de la machine virtuel
- accessible de puis l'ip de la 1ere machine hôte de la hiérarchie(ip_hôte:8888)

**Étapes de démarrage et de fonctionnement :**

- docker build ./ -t rabbitmq_srv
- docker run -p 9999:15672 --name rabbitmq_srv rabbitmq_srv

**DockerFile :**

```
############################## RabbitMQ DockerFile ##########################################################
# ip_adress: 172.17.0.2 / gateway : 172.17.0.1 (mode bridge)
# le serveur rabbitMQ devra ouvrir les ports 15672 et 5672
# accès à l'interface web http://container_ip:15672
## login from localhost (user guest, pass guest)
# création d'un utilisateur admin
## rabbitmqctl add_user admin StrongPassword  (test vm mdp)
## rabbitmqctl set_user_tags admin administrator
#############################################################################################################

# Défininie l'image qui sera build
FROM rabbitmq:latest AS rabbitmq_srv

# exécuter des commandes dans votre conteneur
RUN apt-get update -y
# installation de pyhton
RUN apt-get install -y python3-pip python3-dev build-essential

# Créer un point de montage avec un nom spécifique
# permet d'indiquer quel répertoire vous voulez partager avec l'hôte
# le répertoire de l'hôte qui sera partagé doit être spécifié au lancement du container
#VOLUME /dir_rabbitmq

# WORKDIR : défini le répertoire pour RUN, CMD, ENTRYPOINT, COPY, ADD
# permet de modifier le répertoire courant (équivalent de cd)
# si le dossier n'existe pas alors il sera crée automatiquement
#WORKDIR /dir_rabbitmq

# COPY : copie un nouveau fichier, dossier ou données distante et de les ajouter au système de fichier de <dest>
#COPY rabbitmq_create_user.sh /

# installation du plugins interface web management

#RUN chmod +x /rabbitmq_create_user.sh
#CMD ["/rabbitmq_create_user.sh"]

# permet d'ouvrir les ports nécessaires à rabbitMQ
EXPOSE 15672/tcp
EXPOSE 5672/tcp

RUN rabbitmq-plugins enable rabbitmq_management
#     rabbitmqctl start_app && \
#     rabbitmqctl add_user user test1 && \
#     rabbitmqctl set_user_tags test1 administrator && \
#     rabbitmqctl set_permissions -p / test1 ".*" ".*" ".*"

#ENTRYPOINT ["/rabbitmq_create_user.sh"]

#############################################################################################################
```

**Web services RabbitMQ sur le serveur flask :**

```python
def create_rabbitmq_connection(host):

    # Etablie une conneixon avec le server RabbitMQ
    # ('localhost', 15672, '/', credentials))
    credentials = pika.PlainCredentials('guest', 'guest')
    connection = pika.BlockingConnection(pika.ConnectionParameters(host))
    return connection

def create_queue(host, queue_name):
    print("create called")
    data = {}
    data['state'] = 'NO'
    connection = None

    try:
        # on se connecte au serveur rabbitMQ
        connection = create_rabbitmq_connection(host)
        channel = connection.channel()

        # Create an hello queue sur laquelle notre message sera délivré
        # on peut déclarer la queue n'importe quel nombre de fois, elle ne sera crée qu'une seule fois
        channel.queue_declare(queue=queue_name)

        # On ferme la connexion
        connection.close()

        # on met à jour l'objet JSON
        data['state'] = 'OK'

        return data

    except:
        return data
        # On ferme la connexion
        connection.close()

def send_queue(host, queue_name, msg):

    data = {}
    data['state'] = 'NO'
    connection = None

    try:
        # on se connecte au serveur rabbitMQ
        connection = create_rabbitmq_connection(host)
        channel = connection.channel()

        # Envoie un message à la queue
        # 1param(exchange): échange par default identifié par une chaine vide
        # 2param(routing_key): on indique à quelle queud on souhaite délivré le message (queue hello içi)
        # 3param(body): le corps du message
        channel.basic_publish(exchange='', routing_key=queue_name, body=msg)
        print(" [x] Sent msg="+msg+" to queue= "+queue_name)

        # On ferme la connexion
        connection.close()

        # on met à jour l'objet JSON
        data['state'] = 'OK'

        return data

    except:
        return data
        # On ferme la connexion
        connection.close()
```

```python
def receive_queue(host, queue_name):
    print("receive queue called")
    data = {}
    data['state'] = 'NO'
    connection = None

    try:
        # on se connecte au serveur rabbitMQ
        connection = create_rabbitmq_connection(host)
        channel = connection.channel()

        # on définit un callback qui sera appelé(par pika) à chaque nouveau message inséré dans la file
        # et affichera le contenu du message
        def callback(ch, method, properties, body):
            print(" [x] Received %r" % body)
            # on met à jour l'objet JSON
            data['state'] = 'OK'
            # on convertit les bytes du messages récupérés en string
            data['msg'] = body.decode("utf-8")
            print("data= "+json.dumps(data))
            return data

        #### On ne récupère qu'un seul message de la file ######
        method_frame, header_frame, body = channel.basic_get(queue=queue_name)
        if method_frame.NAME == 'Basic.GetEmpty':
            connection.close()
            # on met à jour l'objet JSON
            data['state'] = 'NO'
            data['msg'] = ''
            return data
        else:
            channel.basic_ack(delivery_tag=method_frame.delivery_tag)
            # on met à jour l'objet JSON
            data['state'] = 'OK'
            data['msg'] = body.decode("utf-8")
            print("mesg returned "+json.dumps(data));
            return data


        #### Callback ####
        # Reçoit les message de la queue lorsqu'il y a en a
        # on indique à RabbitMQ que le callback définit ci dessus doit recevoir les messages de la queue 'hello'
        # on doit bien sur s'assurer que la queue existe avant de s'abonner à ce callback
        #channel.basic_consume(queue=queue_name,
        #                      auto_ack=True,
        #                      on_message_callback=callback)

        # On boucle en attendant les données et appelons le callback quand message reçu
        #print(' [*] Waiting for messages. To exit press CTRL+C')
        #channel.start_consuming()

        # On ferme la connexion
        #connection.close()

    except Exception as e:
        print("data exception")
        # On ferme la connexion
        connection.close()
        return data
        print("receive_queue error= "+e)

def delete_queue(host, queue_name):···
```

# 10. Serveur Flask:

**Quoi ?**

Le serveur web Flask aura pour but d'héberger les web services pour accéder aux files de message RabbitMQ

**Rôles :**

- faire suivre les requêtes de création de queue
- faire suivre les requêtes d'envoie de messages
- faire suivre les requêtes de réception de messages

**Configuration :**

- IP : 172.17.0.3:8081 (2eme container à être lancé)
- Container Ubuntu 18.04
- un forward de port a été configuré sur virtual box du port hôte 8888 vers le 9999 de la machine virtuel

**Dockerfile :**

```
################################ FLASK DockerFile ###################################################################
  # ip_adress: 172.17.0.3 / gateway : 172.17.0.1 (mode bridge)
  # on installe le serveur flask sur la base d'une image Ubuntu
  # et tous ce qui lui est nécessaire
  # serveur web hébergeant les web services
  # Plusieurs informaitons devront lui être communiquée
  # - nom du script.sh lancé au démarrage
  # - addresse ip du serveur rabbitmq
  # hébergera les web services
  ###################################################################################################################

  # serveur démarré sur l'ip du container port 8081
  # Définie l'image utilisé
  FROM ubuntu:18.04 AS flask_srv

  # exécuter des commandes dans votre conteneur
  # mise à jour de la liste de paquets
  RUN apt-get update -y
  # RUN apt-get install -y ping
  # installation de l'utilitaire ping
  RUN apt-get install -y iputils-ping
  # installation de pyhton et pip3
  RUN apt-get install -y python3-pip python3-dev build-essential

  # Créer un point de montage avec un nom spécifique
  # permet d'indiquer quel répertoire vous voulez partager avec l'hôte
  # le répertoire de l'hôte qui sera partagé doit être spécifié au lancement du container
  VOLUME /dir_flask

  # WORKDIR : défini le répertoire pour RUN, CMD, ENTRYPOINT, COPY, ADD
  # permet de modifier le répertoire courant (équivalent de cd)
  # si le dossier n'existe pas alors il sera crée automatiquement
  WORKDIR /dir_flask

  # COPY : copie un nouveau fichier, dossier ou données distante et de les ajouter au système de fichier de <dest>
  COPY flasksrv /dir_flask/flasksrv
  COPY /flask_entry.sh /dir_flask/

  # installation des modules flask, flask-cors et pika
  RUN pip3 install flask
  RUN pip3 install flask-cors
  RUN pip3 install pika

  # ENTRYPOINT: Permet de configurer le container afin qu'il lance un script
  # ENTRYPOINT ["/dir_flask/flask_entry.sh"]
  ENTRYPOINT ["python3"]

  # démarrer le serveur web flask
  CMD ["/dir_flask/flasksrv/run.py"]

  # permet de rendre accessible les ports de puis l'extérieur du container(host)
  EXPOSE 8081

  ###################################################################################################################
```

**Routes du serveur :**

```python
app = Flask(__name__)
# On importe l'ensemble des variables défini dans le fichier config.py3
# To get one variable, tape app.config['MY_VARIABLE']
# app.config.from_object('config')
# app.config['CORS_HEADERS'] = 'Content-Type'
# cors = CORS(app, resources={r"/rabbit/": {"origins": "*"}})
# cors = CORS(app, resources={r'/*': {"origins": '*'}})
# app.config['CORS_HEADERS'] = 'Content-Type'
CORS(app)
# cors = CORS(app)
################ ALL ROUTES #####################################################################

# ROUTE racine qui renvoie l'ensemble des services exposés et l'interface web pour les utiliser
@app.route('/')
def index():
    return render_template('index.html')

# ROUTE /rabbit/ [POST] @param nom_queue
# Créer la queue
@app.route('/rabbit/',methods = ['POST'])
def rabbit():
    print("create called")
    result = request.form
    n = result['nom_queue']
    return json.dumps(zmapp.svc.api_file.create_queue("172.17.0.2", n))

# ROUTE /rabbit/<variable> [POST] @param nom_queue @param2 message
# on définit la route rabbit pour envoyer le message@param à la queue@param
@app.route('/rabbit/<var>', methods=['POST'])
def send_msg(var):
    print("send called")
    #result = request.form
    #n = result['nom_queue']
    #m = result['msg']
    #return zmapp.svc.api_file.send_queue("172.17.0.2", n, m)

# ROUTE /rabbit/<variable> [GET] @param nom_queue
# on définit la route rabbit pour recevoir les message de la queue@param
@app.route('/rabbit/<var>', methods=['GET'])
def receive_msg(var):
    print("receive called")
    n = request.args.get('nom_queue')
    #print("queue_name GET= "+n);
    #print("message récupéré= "+str(zmapp.svc.api_file.receive_queue("172.17.0.2", n)))
    return zmapp.svc.api_file.receive_queue("172.17.0.2", n)

# on définit une route pour les fichier javascripts
@app.route('/js/<path:path>')
def send_js(path):
    return send_from_directory('js', path)

###################################################################################################

if __name__ == "__main__":
    app.run()
```

## 9. GitHub : https://github.com/Enfzifh/

**Quoi ?**

Les dépôts Github serviront de support à l'ensemble des ressources du projet.
Un dépôt a également été crée contenant l'ensemble de mon travail.
Le compte et les dépôts sont donc public.

Nom utilisateur et pass du compte github
saddikiu@gmail.com
Enfzifh
fZ|645gzrg@@_(-

**Rôles :**

- stocker les tâches à faire
- stocker le suivi des tâches effectuées
- stocker les tâches terminées
- stocker les dockerfile des différents containers

**Configuration :**

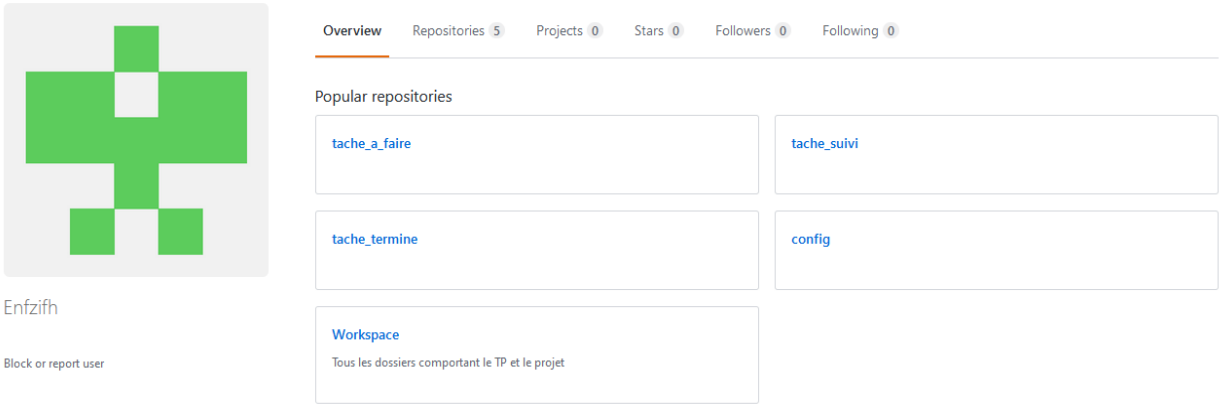
4 dépôts seront crées :

- dépôt tache_a_faire :
  Ce dépôt contiendra les tâches à faire au format JSON

- dépôt tache_suivie :
  Ce dépôt contiendra les parties(job) des tâches faites au format JSON

- dépôt tache_termine :
  Ce dépôt contiendra l'ensemble des tâches terminées(projet) au format JSON

- dépôt config :
  Ce dépôt contient les différents dockerfile



## 10. État des travaux:

**TP 1ere partie :**

→ complet et fonctionnel (dossier env/TP)
→ +interface web : ajax requête pour appeler les webservices RabbitMQ (HS)

**TP 2eme partie(projet) :**

→ incomplet (dossier env/projet)

**Ce qui est fonctionnel/présent:**

- code exécutant (thread)
- code commanditaire (thread)
- serveur flask (container) : build et lancé par le thread exécutant
- serveur rabbitMQ (container) : build et lancé par le thread exécutant
- accès à l'interface web rabbitMQ (forward de port 8888 hôte to 9999 guest VM)
- worker (container) : build par le thread exécutant
- création, upload et download de fichiers sur repos github
- mise en réseau des containers
- création de queue sur le container rabbitMQ depuis le code exécutant
- squelette complet du déroulement du programme en python (MAIN.py)

**Ce qui n'est pas fonctionnel/présent:**

- authentification GITHUB par SSH et échange de la même clé SSH
- mettre les messages(tâches dans la file)
- recevoir les messages de la file
- démarrer un worker par message (car pas de message récupéré)
- implémentation de l'algorithme des n-dames
- génération des dockerfile depuis du code python

**Les incohérences et problèmes que j'ai rencontré :**

- pas de possibilité pour déjuger les requêtes adressés au serveur flask alors que le mode debug était sur true
- gestion des exceptions étranges et parfois non catchées par le « except : »

**Conclusion :**

Les containers sont donc un type de virtualisation que je ne connaissais pas et que j'ai apprécié découvrir et manipuler. J'avais prévu d'utiliser dockercompose pour le lancement d'un seul bloc exécutant mais j'ai du me limiter à de simple dockerfile faute de temps.
Le serveur web Flask est un type de serveur web avec définition de route dont je connaissais le principe.
Le système de file de message était également un nouveau concept pour moi qui m'a fait réfléchir à de nouvelles façons d'imaginer l'implémentation et la gestion de plusieurs idées ; bien que je n'ai pas pu tout expérimenter en terme de configuration, juste en ayant lu la plupart de la documentation j'ai pu me persuader que cette gestion par message était très efficace.
Le langage de développement Python m'a coûté un temps non négligeable, je regrette de ne pas avoir eu le temps de bien revoir, équilibrer et nettoyer mon code (logique, gestion d'exception, manipulation des threads, segment de mémoire partagée que je comptais expérimenter pour mes Workers).
J'ai passé beaucoup de temps sur la modélisation, et au final j'aurai du mieux géré mon temps de travail pour atteindre les principaux objectifs.